Best Practices from the C++ Core Guidelines

Rainer Grimm

Training, Mentoring, and Technology Consulting

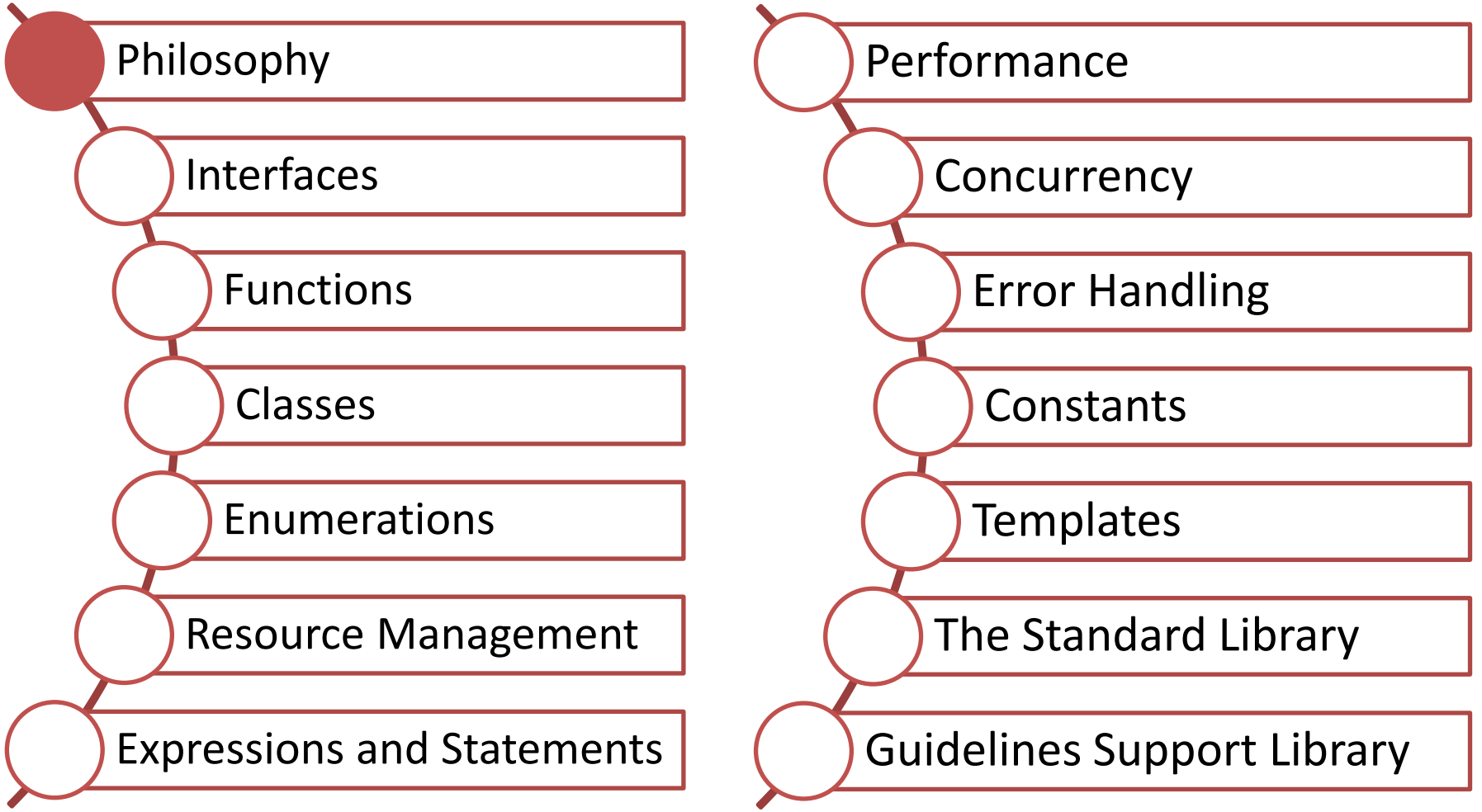
Guidelines

Best Practices for the Usage of C++

- Why do we need guidelines?
  - C++ is a complex language in a complex domain.
  - A new C++ standard is published every three years.
  - C++ is used in safety-critical systems.

Reflect on your coding habits.

C++ Core Guidelines

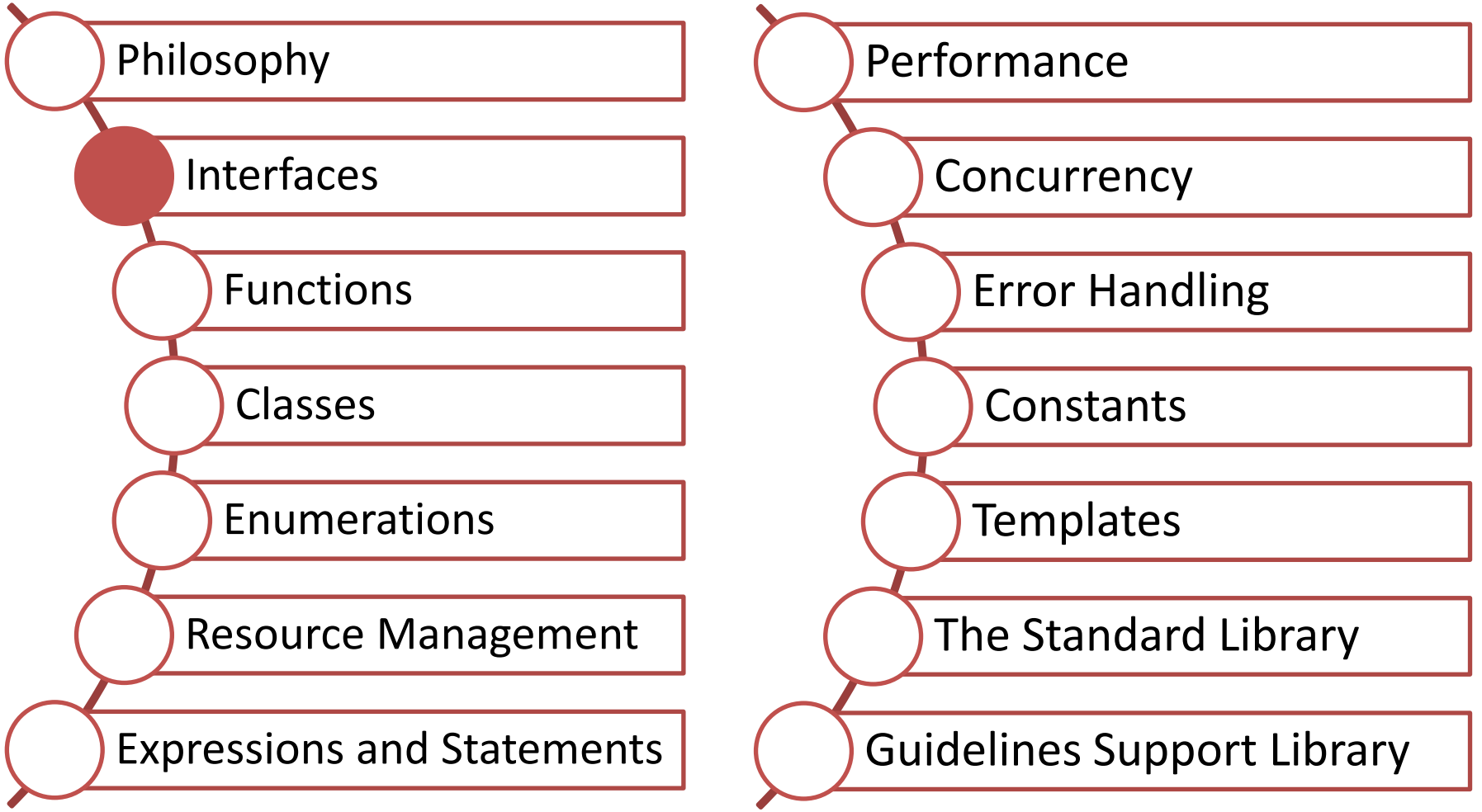


Philosophy

Metarules for the concrete rules.

- Express intent and ideas directly in code.
- Write in ISO Standard C++ and use support libraries and supporting tools.
- A program should be statically type safe. When this is not possible, catch run time errors early.
- Don't waste resources such as space or time.
- Encapsulate messy constructs behind a stable interface.

C++ Core Guidelines



Interfaces

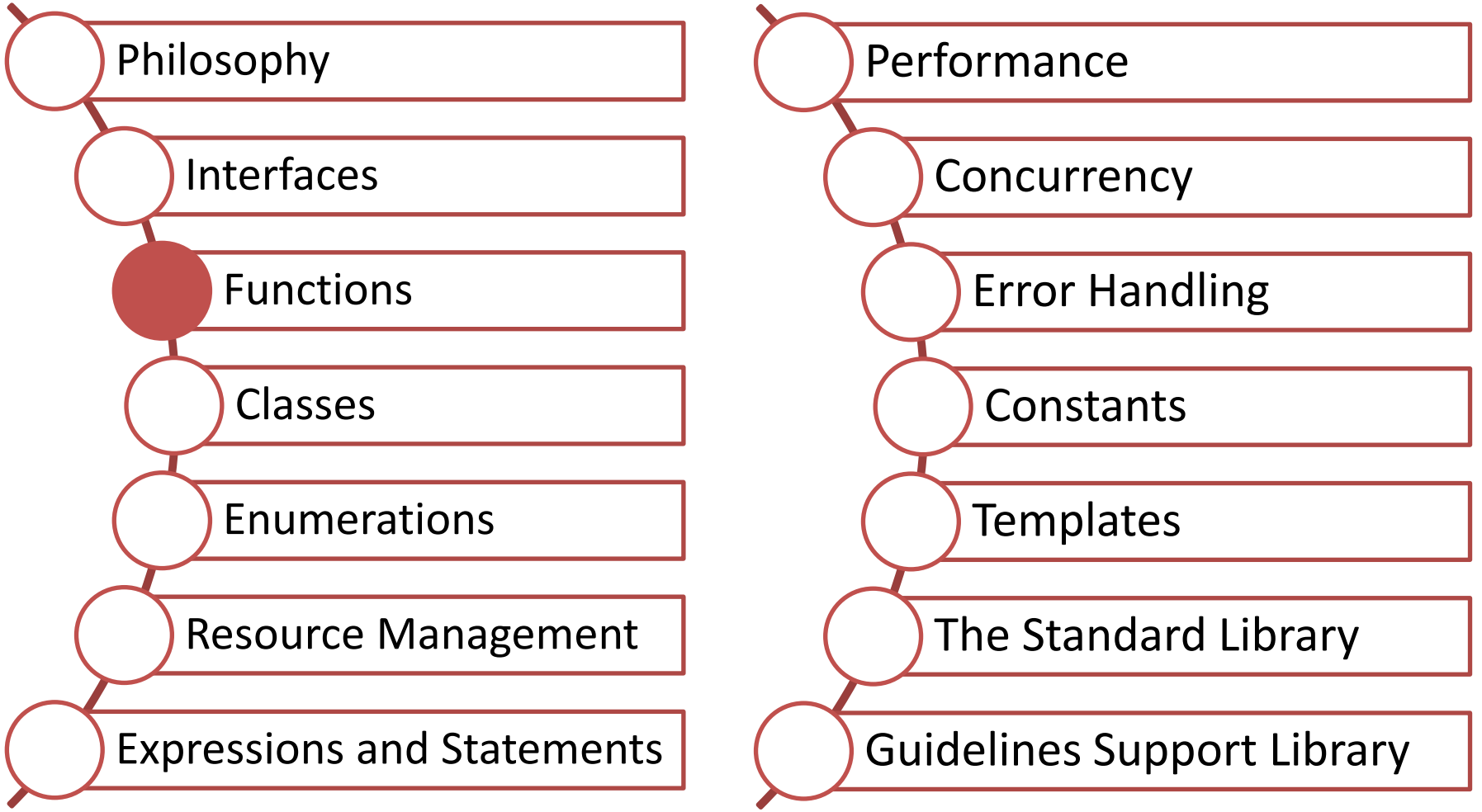
Interfaces should

- be explicit
- be strongly typed
- have a low number of arguments
- separate similar arguments

```
void showRectangle(double a, double b, double c, double d);

void showRectangle(Point top_left, Point bottom_right);
```

C++ Core Guidelines

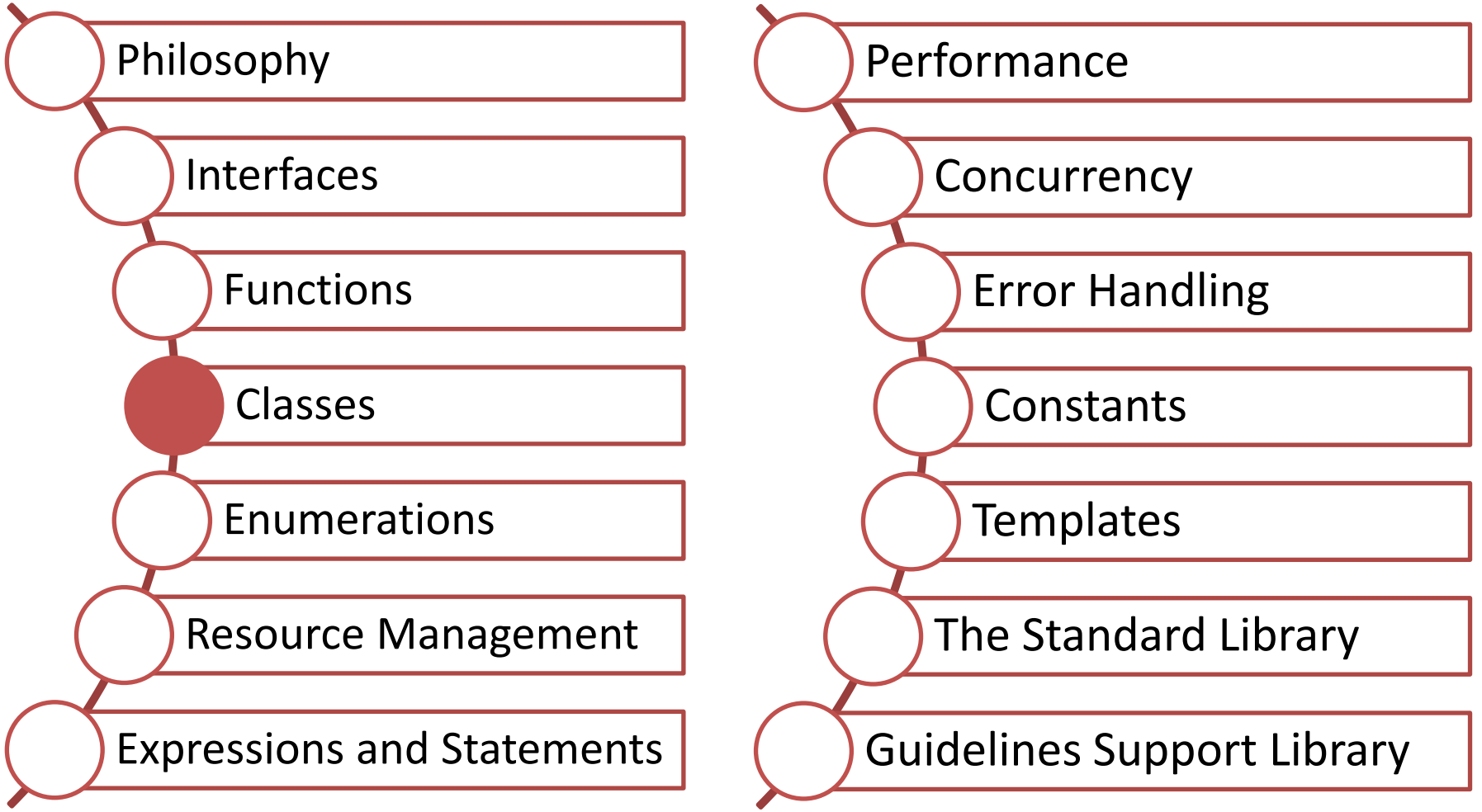


Functions

Ownership semantic of function parameters.

Example	Ownership Semantic
`func(value)`	`func` is an independent owner of the resource
`func(pointer*)`	`func` has borrowed the resource
`func(reference&)`	`func` has borrowed the resource
`func(std::unique_ptr)`	`func` is an independent owner of the resource
`func(std::shared_ptr)`	`func` is a shared owner of the resource

C++ Core Guidelines



Classes and Class Hierarchies

Class hierarchies organize related classes into hierarchical structures.

`class` versus `struct`

- Use a class if it has an invariant
- Establish the invariant in a constructor

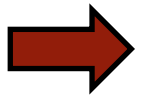
```
struct Point {
    int x;
    int y;
};
```

```
class Date {
 public:
     Date(int yy, Month mm, char dd);
 private:
     int y;
     Month m;
     char d;
};
```

Concrete Types

A concrete type (value type) is not part of a type hierarchy. It can be created on the stack.

A concrete type should be regular.

- Default constructor: `X()`
- Copy constructor: `X(const X&)`
- Copy assignment: `operator = (const X&)`  **Big Six**
- Move constructor: `X(X&&)`
- Move assignment: `operator = (X&&)`
- Destructor: `~(X)`
- Swap operator: `swap(X&, X&)`
- Equality operator: `operator == (const X&)`

Classes and Class Hierarchies

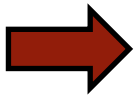
The Big Six

- The compiler can generate them
- You can request a special member function via `default`
- You can delete an automatically generated function via `delete`
- Define all of them or none of them (rule of six or rule of zero)
- Define them consistently
- There are strong dependencies between the big six

Constructor

Don’t define a default constructor that only initializes data members; use member initialization instead

```
struct Widget {
    Widget() = default;
    Widget(int w): width(w) {}
 private:
    int width{640};
};
```



Define the default behavior of each object in the class body. Use explicit constructors for variations of the default behavior.

Conversion Constructor and Operator

Make single-element constructors (conversion constructor) and conversions operators `explicit`.


```
class MyClass{
  public:
    explicit MyClass(A){}     // converting constructor
    explicit operator B(){}   // converting operator
};
```

conversionOperator.cpp
convertingConstructor.cpp

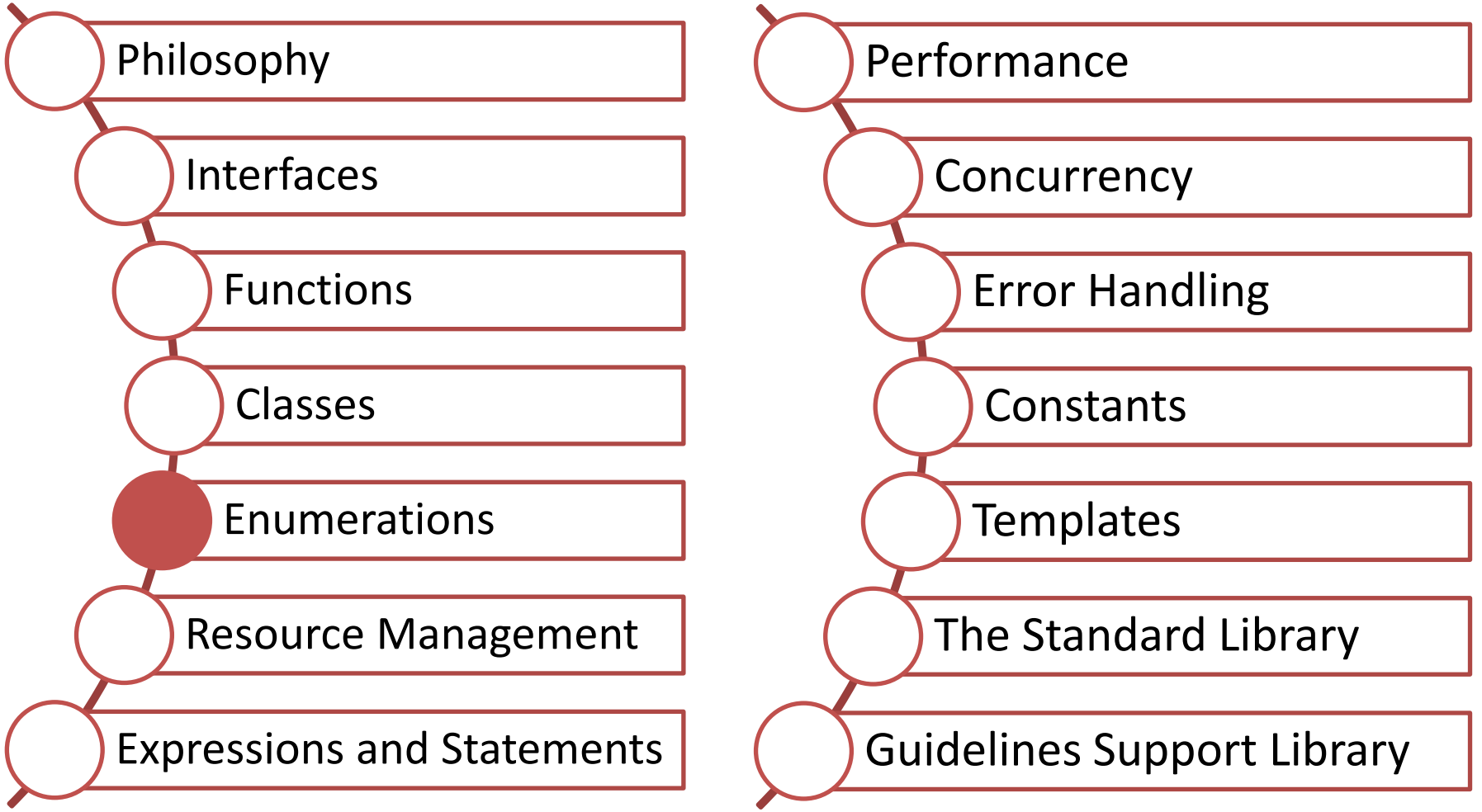
Destructors

- Define a destructor if a class needs an explicit action at object destruction

- A base class destructor should either be public and virtual, or protected and non-virtual
  - `public` and virtual:
    - You can destroy instances of derived classes through a base class pointer or reference
  - `protected` and non-virtual:
    - You cannot destroy instances of derived classes through a base class pointer or reference

- Destructors should not fail ⇨ make them `noexcept`

C++ Core Guidelines



Enumerations

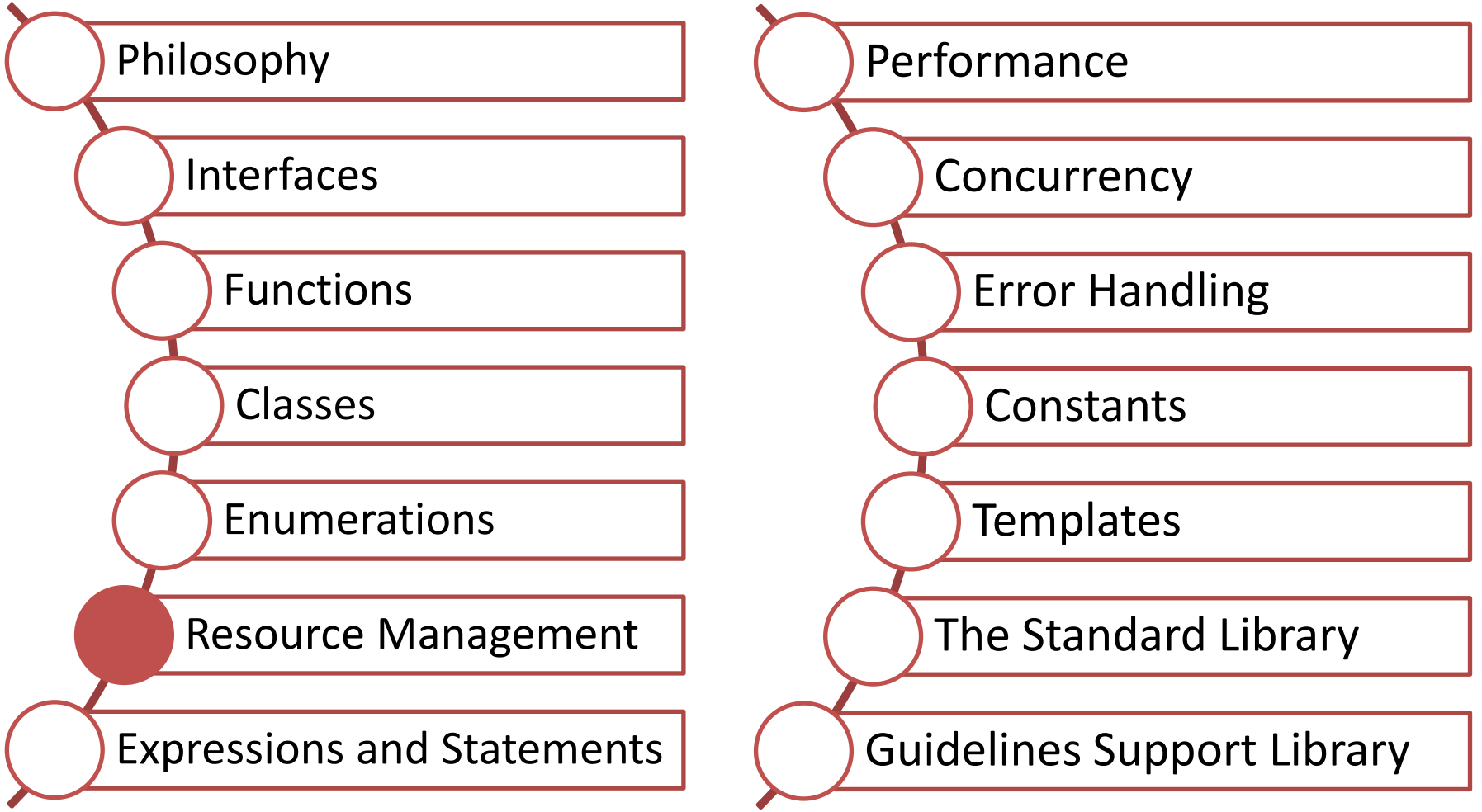
Enumerations are used to define sets of integer values and also a type for such sets of values.

- Use enumerations to represent sets of related named constants
- Prefer enum classes over “plain” enums
- Specify enumerator values only when necessary

```
enum class Day: char {
  jan = 1,
  feb,
  ...
};
```

stronglyTypedEnum.cpp

C++ Core Guidelines



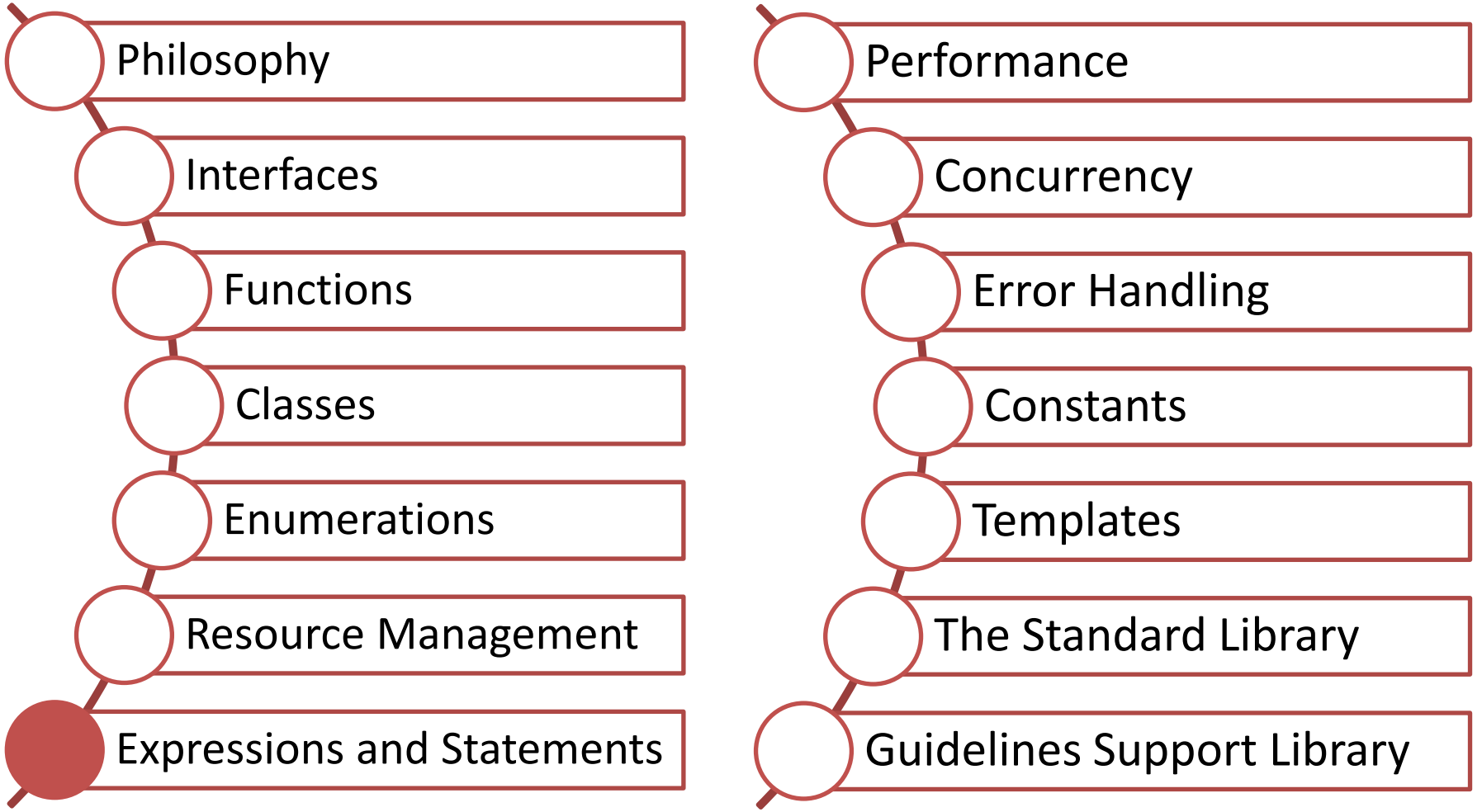
Resource Management: RAII

RAII stands for **R**esource **A**cquisition **I**s **I**nitialization.

- Key idea:
  - Create a local guard object for your resource.
  - The constructor of the guard acquires the resource and the destructor of the guard releases the resource.
  - The C++ run time manages the lifetime of the guard and, therefore, of the resource.
- Implementations
  - Containers of the STL
  - Smart pointers
  - Locks
  - `std::jthread`

raii.cpp

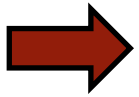
C++ Core Guidelines



Good Names

- Good names are the most important rule for good software.

- Good names should

  - Be self-explanatory.  The shorter the scope, the shorter the name.
  - Don‘t be reused in nested scopes.
  - Should avoid similar-looking names:

```
if (i1 && l1 && ol && o1 && o0 && ol && I0 && l0) surprise();
```

Arithmetic

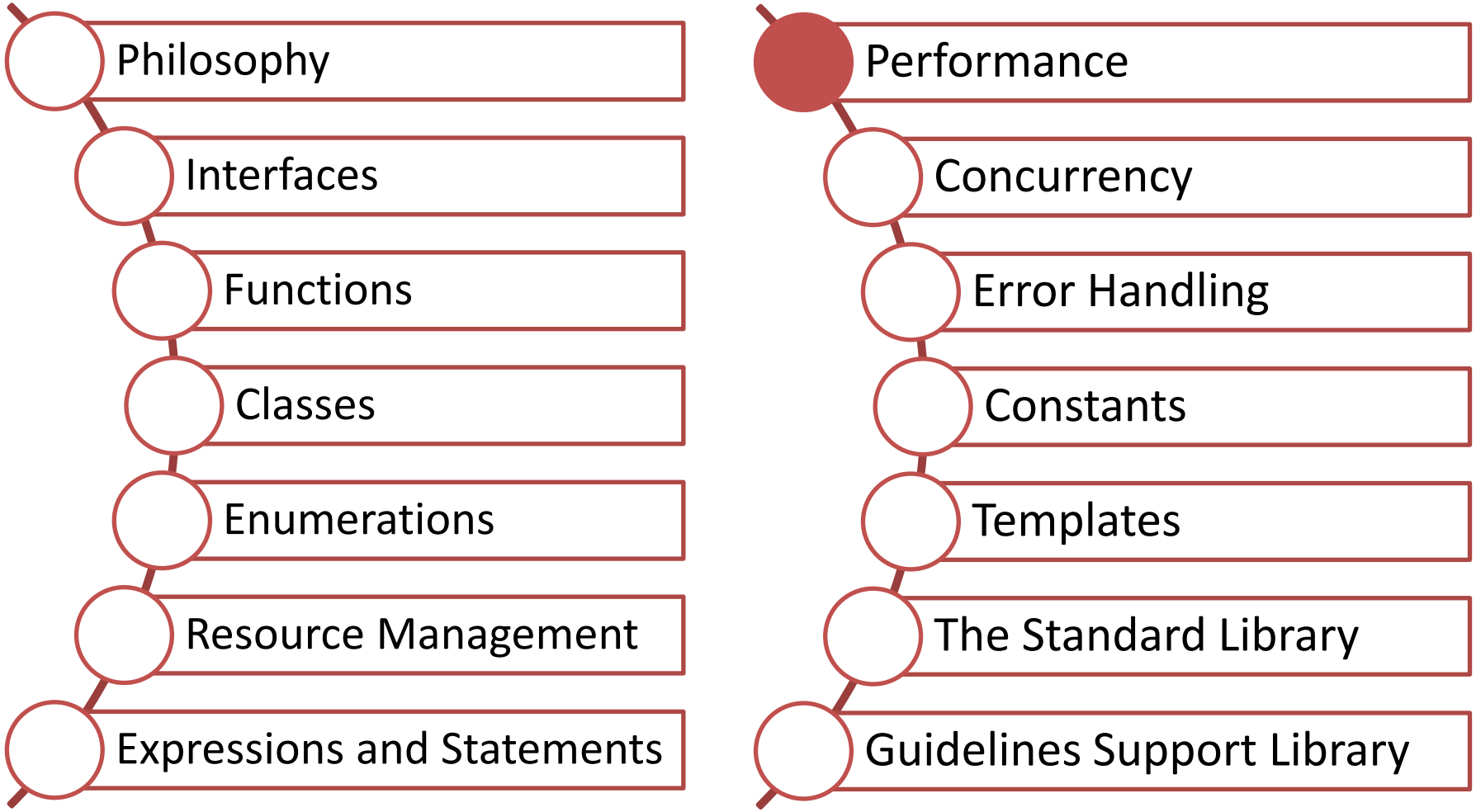
- Don’t mix signed and unsigned arithmetic.

```
#include <iostream>

int main() {
    int x = -3;
    unsigned int y = 7;
    std::cout << x - y << '\n';
    std::cout << x + y << '\n';
    std::cout << x * y << '\n';
    std::cout << x / y << '\n';
}
```

signedUnsigned.cpp

C++ Core Guidelines



Performance

Wrong optimization

- “premature optimization is the root of all evil” (Donald Knuth)

- Rule for optimization
  - Measure with real-world data
  - Versionize your performance test

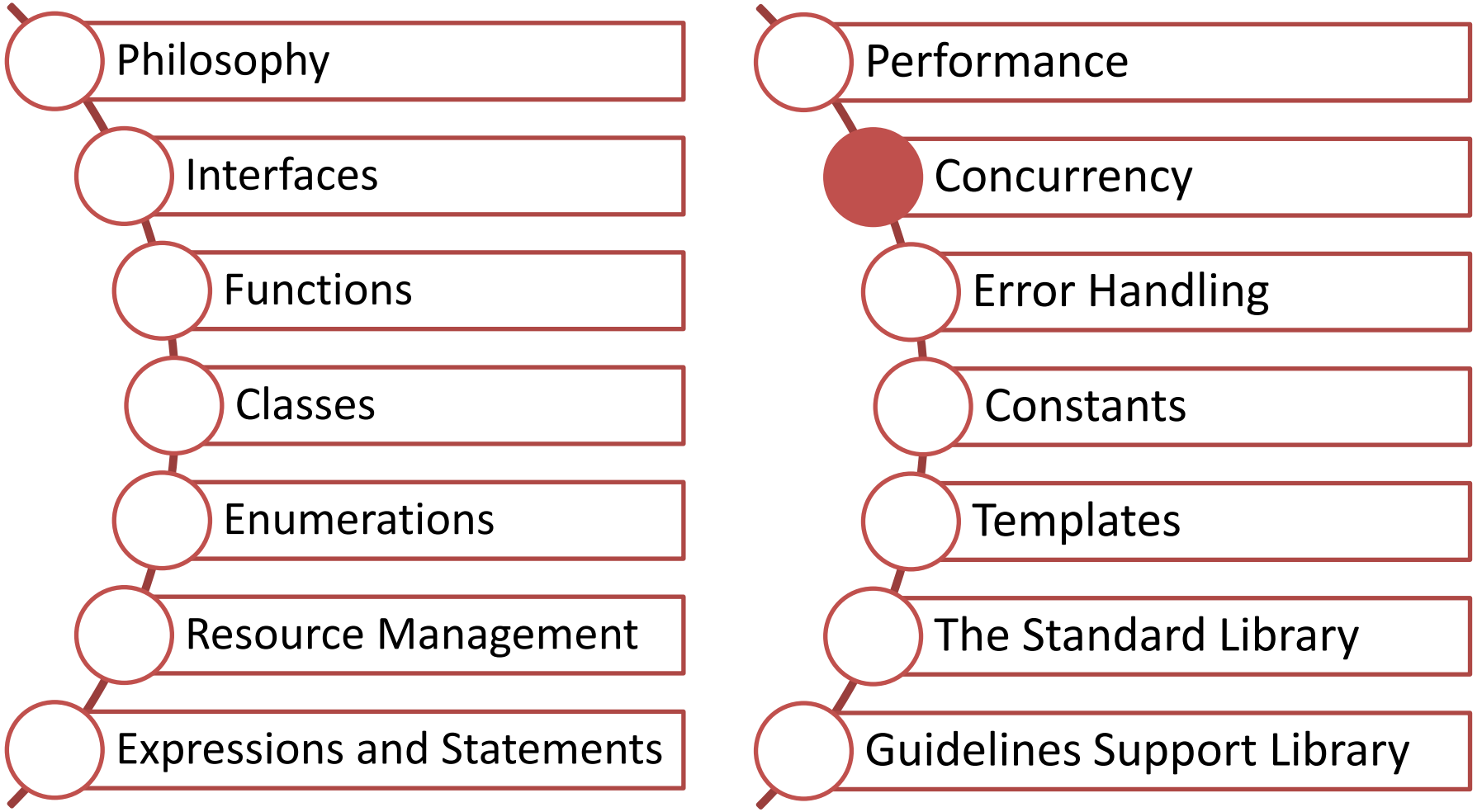
- Importance of measuring
  - Which part of the program is the bottleneck?
  - How fast is good enough for the user?
  - How fast could the program potentially be?

Performance

Enable Optimization

- Use move semantics if possible
- Use `constexpr` if possible
- Rely on the optimizer
  - Write local code
  - Write simple code
  - Give the compiler additional hints (`noexcept, final`)

C++ Core Guidelines



Concurrency and Parallelism

Threads

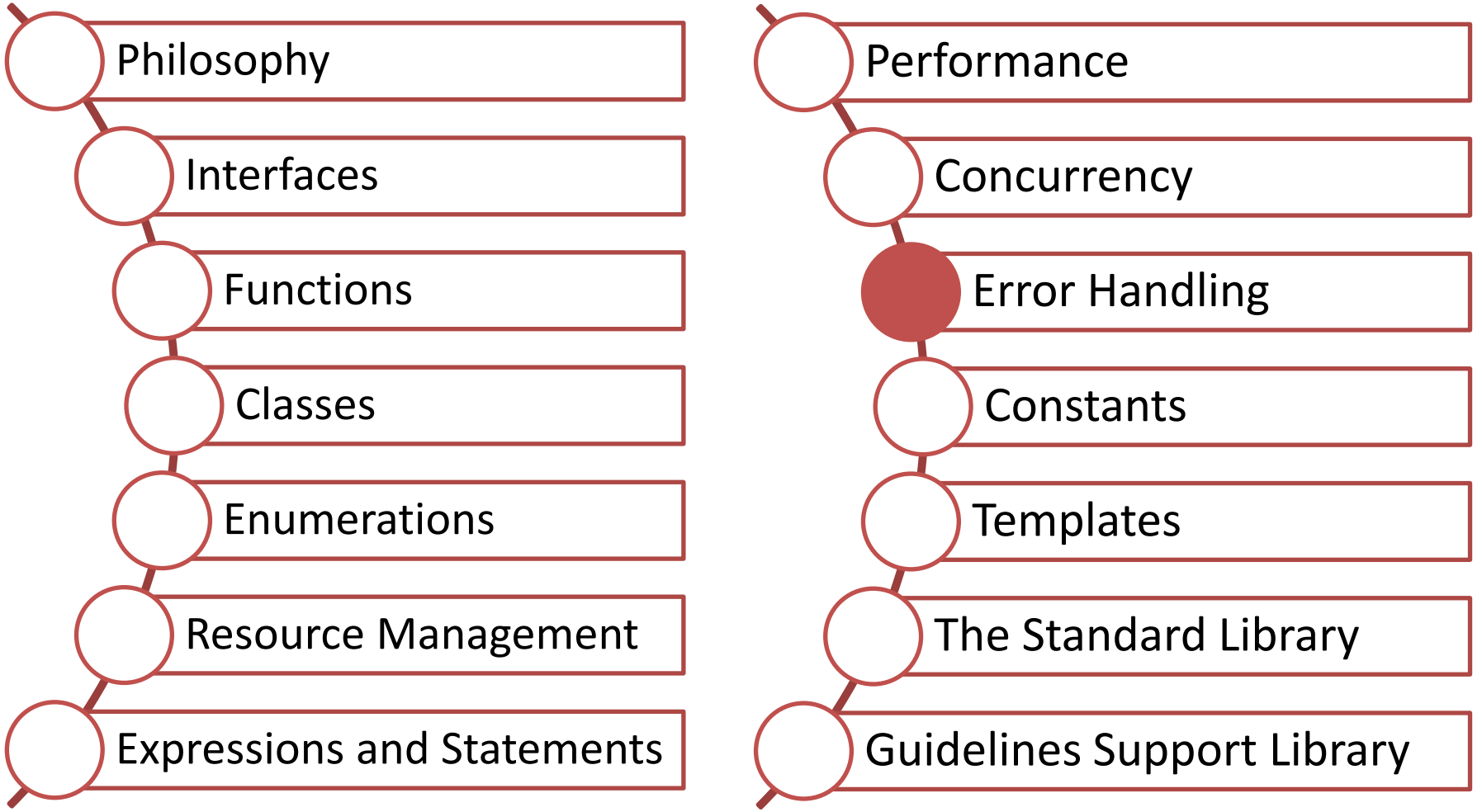
- Prefer `std::jthread` to `std::thread`
- Don’t detach a `thread`
- Pass small amounts of data between threads by value
- To share ownership between unrelated threads use `std::shared_ptr`

threadDetach.cpp

Concurrency and Parallelism

- Use each tool you can get to validate your concurrent code
  - ThreadSanitizer
    - Dynamic code analyzer
    - Part of clang 3.2 and GCC 4.8
    - Compile your program with `-sanitize=thread -g`
  - CppMem
    - Static code analyzer
    - Validates small code snippets, typically including atomics
    - Gives your deep insight into the C++ memory model

C++ Core Guidelines

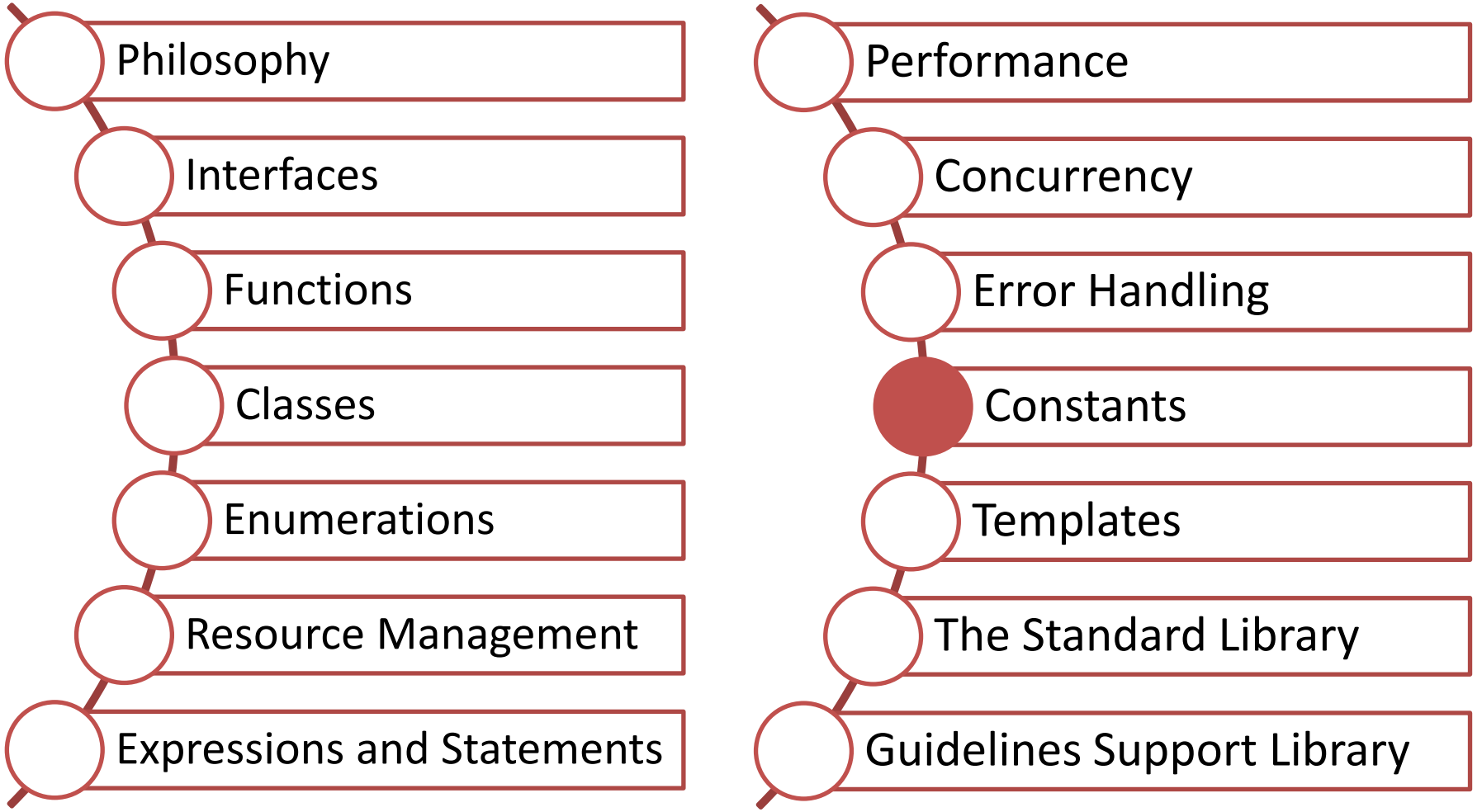


Error Handling

Error handling consists of

- Detect the error
- Transmit information about an error to some handler code
- Preserve the valid state of a program
- Avoid resource leaks

C++ Core Guidelines



Constants and Immutability

- By default, make objects immutable
  - Cannot be a victim of a data race
  - Guarantee that they are initialized in a thread-safe way
  - Distinguish between physical and logical constness of an object

- Casting away `const` from an original `const` object is undefined behavior if you modify it

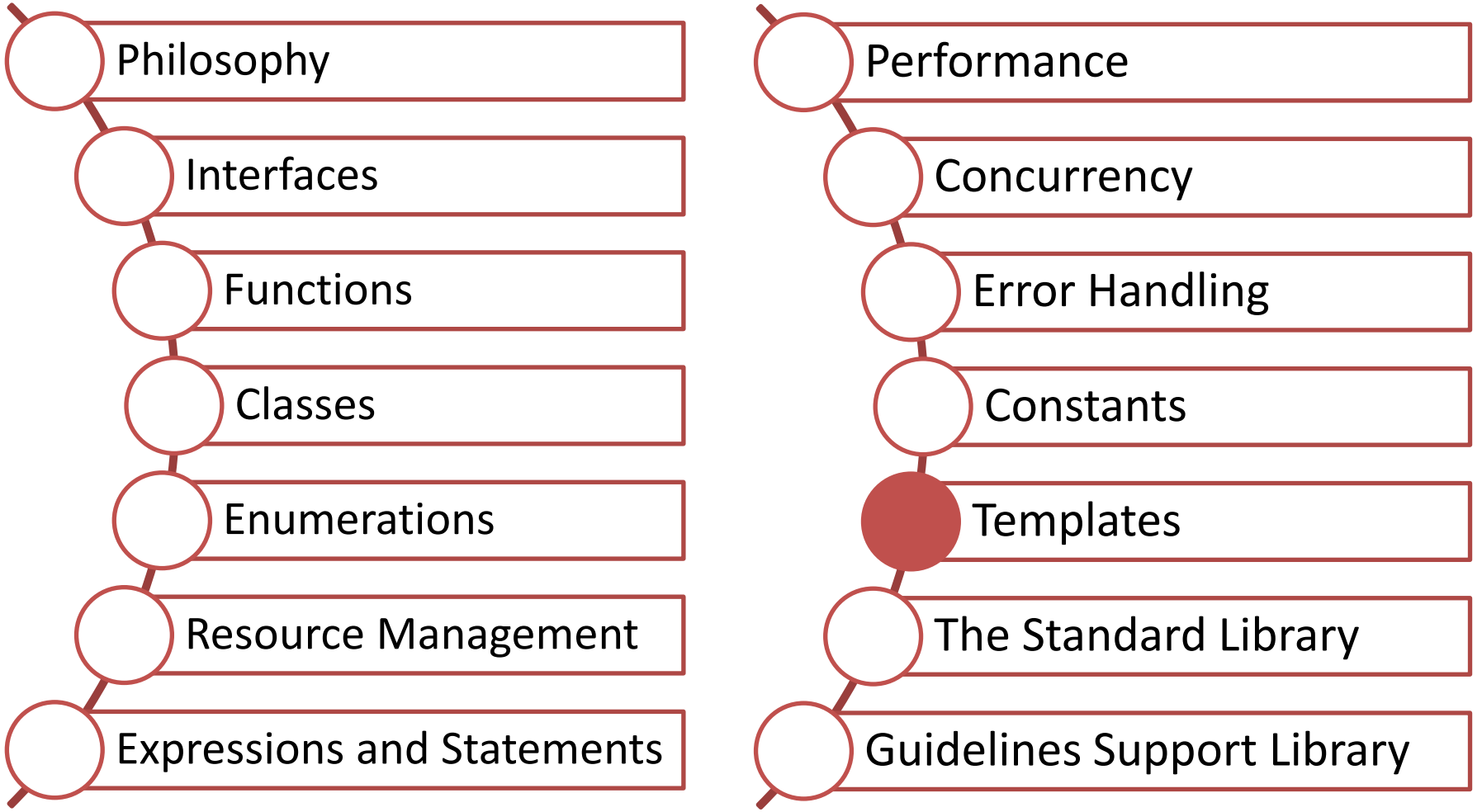
castAwayConst.cpp

Constants and Immutability

- **Physical constness**:
  - The object is `const` and cannot be changed.
- **Logical constness**:
  - The object is `const` but could be changed.

```
struct Immutable{
    mutable std::mutex m;
    int read() const {
        std::lock_guard<std::mutex> lck(m);
        // critical section
        ...
    }
};
```

C++ Core Guidelines



Templates and Generic Programming

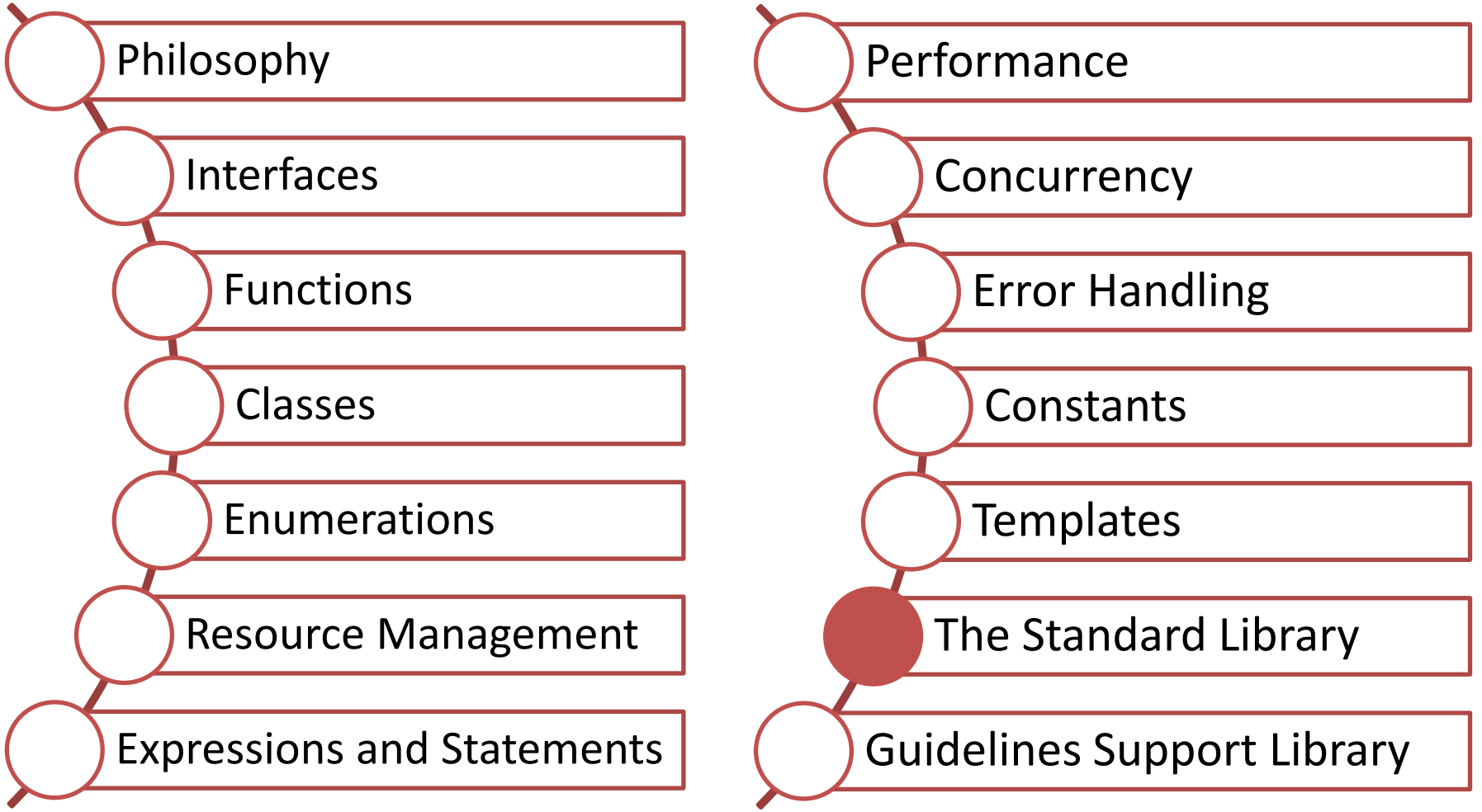
Use

- Use templates to express algorithms that apply to many argument types

Interfaces

- Use function objects (lambdas) to pass operations to algorithms.
- Let the compiler deduce the template arguments.
- Template arguments should be at least `SemiRegular` or `Regular`.

C++ Core Guidelines



`std::array` and `std::vector`

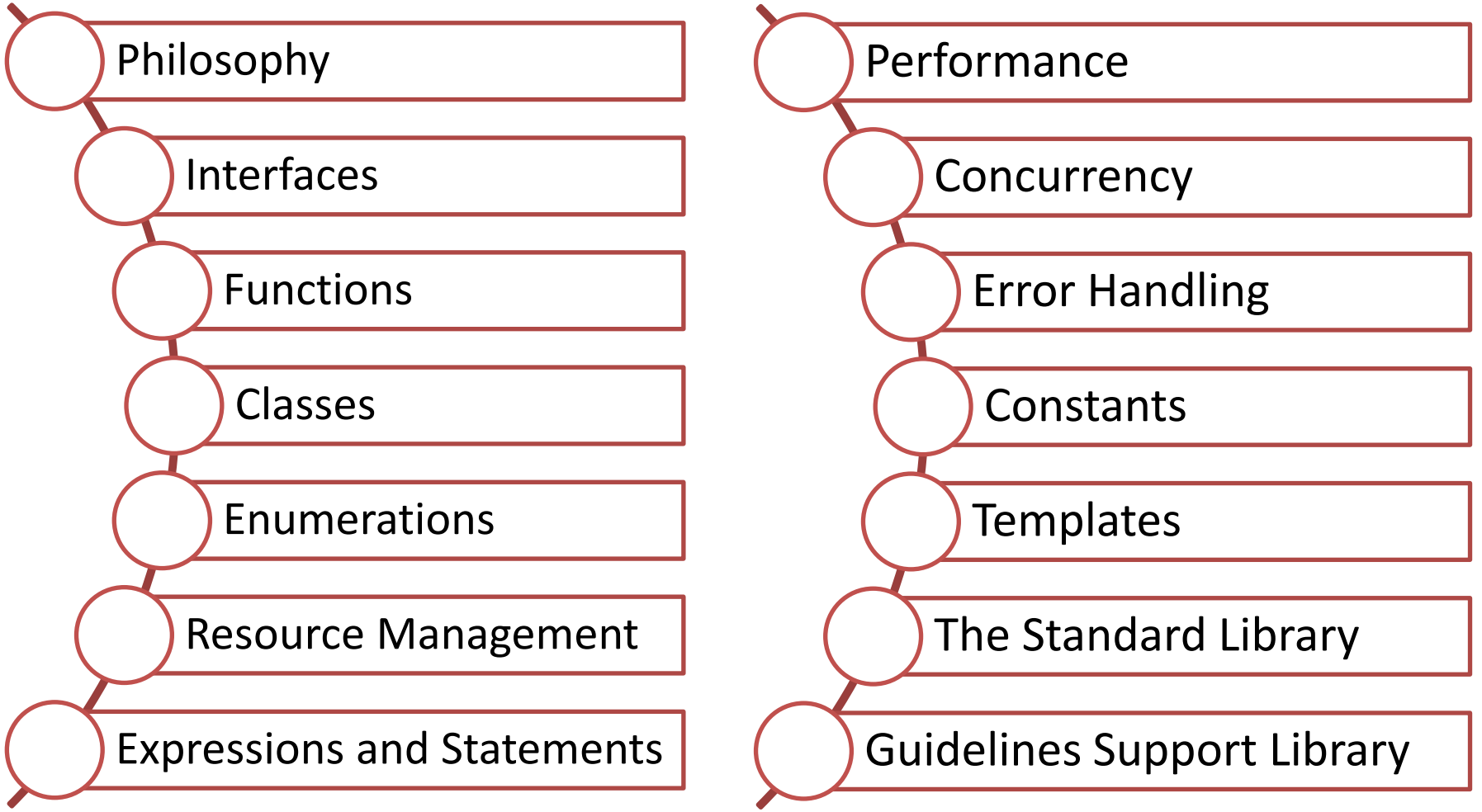
Prefer `std::array` and `std::vector` to a C-array

- The container size is known at compile time and small ⇒ `std::array`
- The container size is not known at compile time or big ⇒ `std::vector`

- `std::vector` and `std::array`
  - know it's size.
  - automatically manage its memory (RAII).
  - allow the protected element access via the `at`-operator.
  - have an ideal memory layout.

`std::array` and `std::vector` should be your first choice for a sequence container.

C++ Core Guidelines



Further Information

[C++ Core Guidelines Explained]

[Beautiful C++]

[Posts about the C++ Core Guidelines on Modernes C++]

[Modernes C++ Training]

[Modernes C++ Mentoring]

Blog: www.ModernesCpp.com
Mentoring: www.ModernesCpp.org

Rainer Grimm

Training, Mentoring, and
Technology Consulting